Blackadder Installation

Introduction

This manual page describes the steps to get, compile and run Blackadder, and all complementary components, like the Topology Manager (TM), in a single machine as well as in a testbed. This manual assumes a clean installation of the latest Debian distribution although it should be accurate for other Debian-like distributions. For the purpose of this manual, we have used Click version 2.0.x and Blackadder release version 0.3.

Quick Start Guide

1. Install the Click modular router
    a. Get Click: *git clone* https://github.com/kohler/click
    b. Compile and install Click
        i. In the *click* directory, run *./configure* or *./configure --disable-linuxmodule* or *./configure --enable-nsclick --enable-blackadder* (for ns-3 bindings)
        ii. (Install Click's dependencies if needed)
        iii. Run *make && sudo make install*
2. Install libraries and applications that Blackadder's components require
    a. Run *sudo apt-get install libtool autoconf automake libigraph0 libigraph0-dev libconfig++8 libconfig++8-dev libtclap-dev*
3. Install the Blackadder core
    a. In the *src* directory of your Blackadder distribution, run *autoconf*
    b. Run either *./configure* or *./configure --disable-linuxmodule* depending on whether you're going to run Blackadder only in user space or also in the kernel
    c. Run *make && sudo make install*
4. Install the Blackadder API library (C++)
    a. In the *lib* directory of your Blackadder distribution, run *autoreconf -fi*
    b. Run *./configure*
    (if you need additional language bindings, see "Installing Blackadder user library")
    c. Run *make && sudo make install*
5. Compile the topology manager (in your TM node)
    a. In the *TopologyManager* directory of your Blackadder distribution, run *make*
6. Compile the deployment tool (in your deployment node)
    a. In the *deployment* directory of your Blackadder distribution, run *make*
7. Deploy Blackadder in your network
    a. Create a network configuration file for your network topology
    (see "Deploying Blackadder")
    b. In the *deployment* directory, run *./deploy -c yournetwork.cfg*
    c. (Re)start Click in your nodes if needed: *click /…/NODELABEL.conf*
    d. Start the topology manager in the TM node: *./tm /…/topology.graphml*
8. Try some of the applications in the *examples* directory

More detailed instructions are given in the following sections.

Compiling and Running Core Components

Note that some of the commands need to be run with root privileges by using, e.g., **sudo**!

Installing Click

Get Click from GitHub: *git clone https://github.com/kohler/click*

Note that although Blackadder v0.3 should work also with the Click 2.0.1 codebase from September 2011, you might experience package installation problems if you choose to use Click 2.0.1 (e.g., *pkg-Makefile* not found). Therefore it's recommended to install a more recent Click version (e.g., from January 2012 or later) instead. If you already have an older version of Click from GitHub, you can get the latest version by running *git pull* in the *click* directory.

You can choose to install Click with or without kernel support. If you don't intend to run Click or Blackadder in the kernel, you can run:

./configure --disable-linuxmodule

Alternatively, configure Click with kernel support (if you need it). We assume that a compatible (or a patched kernel) is already installed. Usually it's enough to run just:

./configure

If the command doesn't succeed, follow the instructions that are printed out. You can also specify custom paths if needed:

./configure --with-linux=/usr/src/linux-headers --prefix=/path/to/binaries --with-linux-map=/boot/System.map-version

Moreover, use additional flags to alter the configuration (e.g. *--enable-multithread, --enable-user-multithread*).

Compile and install Click: (sudo) *make install*

By default many Click packages that Blackadder doesn't need will be compiled and linked with Click, resulting in a large library (and large Click module). To avoid that you can use the *mkminidriver* tool (*http://read.cs.ucla.edu/click/docs/click-mkmindriver*) or manually delete the elements that are not required before compiling Click.

Install with NS3 support

To install Click with NS3 support you must first add Blackadder Elements in Click's Elements (or create a link). Click packages are not supported when running in NS3 mode.

ln -s <BLACKADDER_PREFIX>/src <CLICK>/elements/blackadder
./configure --enable-nsclick --enable-blackadder
make
sudo make install

Installing Blackadder

Before compiling and installing Blackadder, you may need to install autoconf:
apt-get install autoconf

In the *src* folder:

- (optional) run *autoconf*
- run *./configure --with-click=/path/to/binaries –prefix=/path/to/binaries*
- run (sudo) *make install*

If you don't need kernel support, you can run *./configure --disable-linuxmodule* before *make install*.

HTML-based documentation will be created by running *doxygen* in the root Blackadder folder.

File Structure

/path/to/binaries/bin/: all user-space Click-related tools as well as Click executable.
/path/to/binaries /sbin/: all kernel-space tools for starting (*click-install*) and stopping (*click-uninstall*) Click.
/path/to/binaries/lib/: all Click-related libraries and all user (*.uo*) and kernel (*.ko*) objects for the installed packages, like Blackadder.

Running Blackadder

To run Blackadder just run Click with a configuration file describing a valid Blackadder node architecture. Check the sample.conf in ~/src for an example Blackadder configuration. Running this instance will result in a Blackadder node that is capable of Inter-Process Information-centric communication (if you use the NODE_LOCAL strategy in the API). LINK_LOCAL strategy requires at least two nodes and DOMAIN_LOCAL strategy requires the Topology Manager to run.

For user-space run: */path/to/binaries/bin/click configuration.conf*
For kernel-space run: */path/to/binaries/sbin/click-install –u configuration.conf*

Installing Blackadder with NS3

Note: Click must be first compiled with NS3 support.

Download and compile NS3 (tested with ns-allinone-3.15).
To save some time build NS3 by running (in the root ns3 folder):
./build.py --disable-nsc --disable-netanim

Then, copy the "glue" code in NS3 (or create link using ln -s instead of cp -r):
cp -r <BLACKADDER_PREFIX>/ns3/blackadder-model <NS3_PREFIX>/ns-3.15/src/blackadder
cp -r < BLACKADDER_PREFIX>/ns3/blackadder-examples/ < NS3_PREFIX >/ns-3.15/examples/blackadder

cd ns-3.15/
./waf configure --enable-examples --enable-tests --with-nsclick=/path/to/click/source
./waf build

Installing Blackadder user library

First install *libtool* if you don't already have it:

```
$ sudo apt-get install libtool
```

Then go to the *lib* directory.

If needed, you can regenerate the *configure* file and some other files, such as *Makefile.in*s in each subdirectory, by first running *autoreconf.* Note that this requires that *autoconf* and *automake* (and *m4*) have been installed on your system.

```
$ sudo apt-get install autoconf
$ sudo apt-get install automake
```

```
$ autoreconf –fi
```

After the optional *autoreconf* step, run *./configure*. The default installation locations are */usr/local/include* and */usr/local/lib*, but the */usr/local* prefix can be changed by giving a different path with the *--prefix* parameter (e.g.: *--prefix=/path/to*). Also other parameters can be given; run *./configure --help* for more information.

```
$ ./configure
```

Then compile the library and install the library and header files:

```
$ make
$ sudo make install
```

Both a shared (*libblackadder.la* and *libblackadder.so.**) and a static library (*libblackadder.a*) are generated. The library is linked with applications by specifying *-lblackadder* as a linker option to (e.g.) *g++*. Also *-lpthread* is normally needed. In case you need to do static linking, add *-static* as an option (and if you want to completely skip installing the library in this case, you can give the *lib/.libs* directory as an *-L* option when you compile the program).

The header files that are normally used in C++ programs are *blackadder.hpp* (that implements blocking event handling) and *nb_blackadder.hpp* (for non-blocking event handling).

Installing SWIG Bindings for Python, Ruby and Java

By default, only the C++ API is included when the library is compiled and installed (see above). Bindings for "higher level" languages, such as Python, Ruby and Java, can be generated with SWIG (*Simplified Wrapper and Interface Generator*) and installed along with the library. Support for Ruby and Java should be considered experimental.

Begin by installing SWIG if you don't already have it:

```
sudo apt-get install swig
```

You also need to have additional language-specific development files (headers) installed as a prerequisite to compiling the bindings. You can install these packages with *apt-get* as usual. For Python, the *python-dev* package is needed. For Ruby, *ruby-dev* is required. For Java, we recommend *openjdk-6-jdk*, but the bindings work with *sun-java6-jdk* as well (if *jni.h* for both is found in the

system, the one for OpenJDK is used by default). In order to install support for Python, Ruby and Java, run:

```
sudo apt-get install python-dev ruby-dev openjdk-6-jdk
```

Enable additional languages by passing one or more *--enable-LANGUAGE* options to the *configure* script in the *lib* directory. Run *./configure --help* to see which languages are supported. Don't forget to include any other parameters you need (e.g., *--prefix*). After this, run or re-run *make* and *make install* to install everything according to the configuration. Example:

```
$ ./configure --enable-python --enable-ruby --enable-java
$ make
$ sudo make install
```

In order to run applications that use the Blackadder library, you may need to specify the location of the library to the compiler and/or interpreter. For Python, set the *PYTHONPATH* environment variable to point to the installation directory (usually the *site-packages* subdirectory). For Ruby, set the *RUBYLIB* variable. For Java, set the *CLASSPATH* variable when compiling and running the program, and also specify the *-Djava.library.path=...* parameter to *java* when running it (even if you installed the library into */usr/local/lib*).

Note that the Python bindings only work with Python 2.x. If you have multiple versions of Python 2.x installed and want to specify which one you want to use with Blackadder, do the following:

1. Edit *lib/configure.ac* and change the *AM_PATH_PYTHON* line to, e.g., *AM_PATH_PYTHON(2.6)*
2. Rerun *autoreconf -i*, then *./configure --enable-python* (etc.), and finally (sudo) *make install*

Installing the C bindings

Similarly as for the SWIG-based bindings, first run *./configure --enable-c* and then *make install* in the *lib* directory. This installs *libblackadder_c.** and *blackadder_c.h*.

```
$ ./configure --enable-c
$ make
$ sudo make install
```

Installing the Java Bindings

An alternative to the SWIG-based Java bindings is located in *{blackadder-dir}/lib/bindings/java-binding*. This particular component offers a slight improvement in performance due to reduced buffer copies between the Blackadder library and JVM when sending and receiving data packets, as well as a few classes that provide an object-oriented abstraction to the network functionalities. The bindings have been tested mainly with OpenJDK (which by default is assumed to be installed on the system), but the native interface should be compatible with Sun's Java VM as well (and can be compiled with Sun's *jni.h*).

To use this binding, first you need to build and install the Blackadder user library, following the instructions above. To build the Java sources, you need to have ant installed. In Debian/Ubuntu distributions, you can install ant via

```
sudo apt-get install ant
```

You'll also need to download the Apache Commons Codec library from http://archive.apache.org/dist/commons/codec/binaries/commons-codec-1.4-bin.zip, extract the commons-codec-1.4.jar file and copy it to the *java-binding/lib* directory. (Later versions of the library might also work.)

The Java binding consists of two parts: (i) the native part and (ii) Java code. First, you need to compile and build the native part. Go to the *{blackadder-dir}/lib/bindings/java-binding/jni* directory. Update the *Makefile* and make sure that the JNI_LOC variable points to the location of you JVM header files. If you are unaware of the location of the header files, you may find them by running

```
(sudo) find / -name jni.h
```

Save the Makefile and run *make*. This will produce the object file named *eu_pursuit_client_BlackadderWrapper.so*. Remember the location of this file as you will have to give it as input in your java applications. Once you have build the JNI part, go to *{blackadder-dir}/lib/bindings/java-binding/* and run "*ant*" to build the java sources. Classes are copied to the *bin* directory. By running "*ant dist*", you will create the respective .jar file, copied in the *dist* subdirectory. You can import this jar as a library to java applications.

If you want to change/update the Java binding itself, you can directly import it to Eclipse. The java-binding directory is an Eclipse project folder and should be easily imported to the IDE via "File → Import → Existing Projects into Workspace."

Setting Up and Running the Topology Manager

The Topology Manager is a normal C++ application that accesses the network using the libraries produced by following the aforementioned steps. In order to compile the Topology Manager, iGraph and libconfig++ libraries are required.
Install both by running: *(sudo) apt-get install libigraph0 libigraph0-dev*
Compile the Topology Manager by running make in ~/TopologyManager folder. The executable is called *tm*.
Run the Topology Manager using: ./tm topology.graphml, topology.graphml is a standard format for defining a graph (a network topology in this case). This file is produced automatically by the deployment application, described below.

Deploying Blackadder

The deployment utility is an external tool that reads a configuration file, the syntax of which is described below, and produces Click configuration files for all nodes in a testbed. It copies these files to the destination nodes and run Click (in the mode defined in the configuration file). It also creates the .graphml file and runs the Topology Manager in the node declared in the configuration using this file that is copied there.
The deployment utility requires iGraph, libconfig++ and libtclap libraries. Install them by running:
(sudo) apt-get install libigraph0 libigraph0-dev libconfig++8 libconfig++8-dev libtclap-dev

Configuration Options

Mandatory Global Options

BLACKADDER_ID_LENGTH: the length of Scope IDs and Information IDs supported by Blackadder. Currently this parameter has to be configured separately in Blackadder at compile time (PURSUIT_ID_LEN in *src/helper.hh*, *lib/blackadder_defs.h* and, if one use the ns-3 bindings, *ns3/blackadder-model/model/service-model.h*).

LIPSIN_ID_LENGTH: the length of LinkIDs, internal Link IDs, and LIPSIN identifiers in bytes.

CLICK_HOME: the absolute path where Click is installed.

WRITE_CONF: the absolute path where the deployment utility will remotely copy the Click/Blackadder configurations in each Blackadder node. The same is going to be used to remotely copy the produced topology.graphML file at the network node that will run the Topology Manager.

USER: The username of the user that will be used when ssh-ing network nodes (for retrieving mac addresses and running Click) and copying configuration files.

SUDO: True if the deployment utility will use *sudo* when remotely executing commands to network nodes.

OVERLAY_MODE: The mode in which Blackadder will run at the testbed. Currently, Blackadder can run on top of Ethernet ("mac") or Raw IP Sockets ("ip").

Defining a Network

A network is defined in the configuration file as follows:

```
network = {
        nodes = (
                { ….node1
                },
                { …node2
                }
                );
        };
```

A configuration file can currently store a single network. The abovementioned global parameters are valid for the whole network (including all nodes and all connections).

Defining a Network Node

A network node is defined in the configuration file as follows:

```
{
    testbed_ip = "";
    running_mode = "";
    label = "";
    role = ["",""];
    connections = (
    {
       … connection 1
    },
    {
       … connection 2
    }
    );
  }
```

testbed_ip: The IP address (in dotted decimal format) to which the deployment utility will copy the configuration files and remotely execute commands.
running_mode: The mode in which Blackadder will run in this node. Use "user" for user-space (as user-space process) or "kernel" for kernel space (as a Linux module). Note that in a testbed some nodes may run in user-space while others run in kernel-space.
label: The Label of that network node. The label is used when sending requests to the Rendezvous Node. The Topology Manager also keeps track of the nodes in the network using their labels. The size of the label must be *BLACKADDER_ID_LENGTH* bytes.
role: if omitted or role[] then the network node has no special functionality. Use role["RV","TM"] if the node is the Rendezvous Node and the Topology Manager or use the above keywords separately to place the (extra) functionalities to different nodes.

Defining a Network Connection

A network connection is always unidirectional and is defined within the context of a network node as follows:

```
{
   to = "00000002";
   src_ip = "10.0.1.18";
   dst_ip = "10.0.1.19";
}
```

to: the destination node (its label)

If the network runs on top of IP:
src_ip: the source IP address that will be used when sending to the Raw IP Socket.
dst_ip: the destination IP address that will be used when sending to the Raw IP Socket.

If the network runs on top of Ethernet:
src_if: the network interface from which publications will be sent. The deployment will use this and remotely acquire the respective MAC address. E.g. use "eth0" or "tap0" and Blackadder will use a physical interface or a virtual one (probably over a VPN), respectively.
dst_if: the network interface to which publications will be sent. The deployment will use this and remotely acquire the respective MAC address.
src_mac: the MAC address of the source network interface. This parameter is optional and if given, the deployment script will not acquire the MAC address remotely. Helpful when the network contains nodes that will not be online at the time the deployment utility will run.
dst_mac: the MAC address of the destination network interface. This parameter is optional and if given, the deployment script will not acquire the MAC address remotely.

For the exact syntax check the sample_topology.cfg file in the deployment folder as well as the configuration file grammar in http://www.hyperrealm.com/libconfig/.

Multilayer-support in the Deployment Tool

Multilayer (ml) support is an additional feature of the original deployment tool that allows generation of multilayer (packet and optical overlay) topology configurations. When multilayer mode is set in the input configuration file, corresponding Click configuration files for all the nodes in

a testbed are produced, and the configuration files are transferred to packet layer nodes (in which Click is also started).

Multilayer Configuration File

The ml configuration file has a modified layout from the original one, as it organizes nodes into groups according to their layer, e.g. *packet layer (pl)* and *overlay layer (ol)*, where each node has a new parameter (in addition to the standard ones) named *'type'* that defines the node type (i.e. packet or overlay node). It also introduces an additional overlay mode named *'mac_ml'*, in addition to *'mac'* and *'ip'* modes.

Defining a Multilayer Network

An ml-network is defined in the configuration file as follows:

```
network = {
        ol_nodes = (
                { …node1
                },
                { …node2
                }
                );

        pl_nodes = (
                { …node1
                },
                { …node2
                }
                );
        };
```

Like the original configuration file, an ml configuration can currently define only a single network (domain). The global parameters are valid for the whole network (including all nodes and all connections).

Defining a Multilayer Network Node

A network node is defined in the ml configuration file as follows:

```
{
    testbed_ip = “”;
    running_mode = "";
    label = "";
    role = [“”,””];
    type =[“”];
    connections = (
    {
       … connection 1
    },
    {
       … connection 2
    }
    );
  }
```

type: The type of that network node, used to distinguish between nodes that belong to one layer and nodes that belongs to the other layer for the purpose of performing different, layer-based, processes. The current supported types are:

- ["PN"]: packet layer node
- ["ON"]: overlay layer node

Defining a Multilayer Network Connection

As in other overlay modes an ml connection is a unidirectional one. However, due to the multilayer nature of the network topology, three different types of connections can be defined, depending on the source and sink of the connection. Connections are defined as follows:

- For an overlay-to-overlay connection (oo), the definition is:

```
{
   to = "00000002";
   out_pt = "1";              // output_port (i.e. signal departing from the node)
   in_pt  = "2";              // input_port (i.e. signal arriving to the node)
 }
```

- For a packet-to-packet connection (pp), the definition is:

```
{
  to = "00000002";
  src_if = "eth0";
  dst_if = "eth1";
}
```

- For a packet-to-overlay or overlay-to-packet (i.e. cross layer 'cl' connection), the definition is:

```
{
  to = "00000002";
  src_if = "eth0";
  in_pt  = "1";
}
```

Note that packet layer connections should run on top of Ethernet, to facilitate the direct link connectivity between the packet and overlay layers.

out_pt: The output port of an "ON" , where the data departs from the node a cross the link towards its destination. out_pt is defined for optical overlay node (OXC) by a number that identifies the port index in the node.

in_pt: The input port of an "ON", which connects to the arrival end of a link. Similar to out_pt, it is defined by its index number in the node.

Section '*Defining a Network Connection*' above provides definitions for 'src_if', 'dst_if', etc.

For the exact syntax check the *ml_configuration.cfg* file in the *deployment* folder.

NS3 Support

Defining an NS3 Network

An NS3 network is defined in the configuration file as follows:

```
network = {
        nodes = (
                { ….node1
                },
                { …node2
                }
```

```
        );
    };
```

A configuration file can currently store a single network. The abovementioned global parameters are valid for the whole network (including all nodes and all connections).

Defining an NS3 Network Node

A network node is defined in the configuration file as follows:

```
{
    label = "";
    role = ["","",""];
    connections = (
    {
      … connection 1
    },
    {
      … connection 2
    }
    );
    applications = (
    {
      … application 1
    },
    {
      … application 2
    }
    );
  }
```

label: The Label of that network node. The label is used when sending requests to the Rendezvous Node. The Topology Manager also keeps track of the nodes in the network using their labels. The size of the label must be *BLACKADDER_ID_LENGTH* bytes.

role: if omitted or role[] then the network node has no special functionality. Use role["RV","TM"] if the node is the Rendezvous Node and the Topology Manager or use the above keywords separately to place the (extra) functionalities to different nodes.

Defining an NS3 Network Connection

A network connection is always unidirectional and is defined within the context of a network node as follows:

```
    {
      to = "00000002";
      Mtu = 1500;
      DataRate = "100Mbps";
      Delay = "10ms";
    }
```

to: the destination node (its label)

Mtu: The simulated MTU of this link

DataRate: The simulated Data Rate of this link

Delay: The simulated propagation delay of this link

Defining an NS3 Application

This block defines a set of NS3 applications that will be simulated as running in this node. Note that the name of the application **must be the same** as the C++ class defining the application in NS3.

```
{
    name = "Subscriber";
    start = "2.34";
    stop = "14.87";
}
```

name: the name of the NS3 application (same as the C++ definition)
start: The time in seconds when this application will start
stop: The time in seconds when this application will stop

Running the Deployment Utility

The Deployment Utility accepts flags to perform a number of different operations. Usage:

```
./deploy  [-t <string>] [-d <string>] -c <string> [--nostart] [--nocopy]
          [--nodiscover] [-m] [-a] [--] [--version] [-h] [-s]
```

Where:

-t <string>, --tgzfile <string>
.tar.gz file that gets transferred and extracted at USER home folders on all experiment targets. The experiment targets are read from the -c <file>.

-d <string>, --experimentfile <string>
Experiment description deployment file that contains information to remotely deploy applications and locally collect their STDOUT (Experimental Feature).

-c <string>, --configfile <string>
(required field) Configuration file that contains the node descriptions (graph attributes) OR describes connections (i.e. a graph) as well (when -a is not used).

--nostart
Don't start/stop Click in the nodes.

--nocopy
Don't copy files to nodes.

--nodiscover
Don't auto-discover MAC addresses.

-m, --montoolstub
Enable Java monitor tool stub in the Click configuration files. This injects counters in Click configs that are inspected at runtime via port 55555. It is automatically ommited when kernel versions are described in the configuration files.

-a, --auto
Enable graph autogeneration - autogenerated.cfg and edgevertices.cfg files are emitted in the WRITE_CONF folder. The former contains the autogenerated graph to repeat the experiment and the latter the leaf nodes of that graph.

-h, --help
Displays usage information and exits.

-s, --simulate
Instead of deploying in a real network, the deployment tool will create all necessary click configuration files, the topology file and NS3 simulation (C++) code.

Deploying Blackadder on PlanetLab

Software Versions

Until recently (September 2011), the Click modular router did not support properly the PlanetLab Sliver port registration mechanism, but Click 2.0.1 and later versions support this. If you are not sure about the version you have you can check the developers' comment that is located in the top of the *elements/userlevel/rawsocket.cc* file in your Click distribution. If it has a note from 2011 about the port registration mechanism update then your Click distribution is up-to-date.

Building Instructions for PlanetLab

Building Blackadder

PlanetLab allocates slivers on PlanetLab nodes for your account. You can log in to these nodes using passwordless SSH with your user account name. Your Click binary on PlanetLab can only be executed on your account home folder of each allocated sliver, e.g: */home/certhple_purs1*.

So when you build Click and Blackadder for the PlanetLab target, create the same folder pattern on your development host and use the appropriate --prefix command when configuring Click and Blackadder. In our example we would create */home/certhple_purs1* on our development host and add *--prefix=/home/certhple_purs1* when executing the *configure* script.

Building Blackadder TM

PlanetLab nodes have the Fedora Core 8 Linux distribution with the *yum* package manager. Blackadder's C++-based TM will need *igraph* that is available as a package and can be installed with yum on the node that you intend to use as the TM node.

Note: PlanetLab slivers are resource constrained. A TM that executes on a sliver can easily become the performance bottleneck, especially for large-scale experiments. It is advised that te TM runs on a dedicated server machine in our own network. All you need is a publicly accessible IP address and firewall configuration for an open UDP port.

Deployment Utility Usage for PlanetLab

Blackadder's universal deployment tool has extended support for PlaneLab. It can do the following:

i) Retrieve the allocated PlanetLab nodes directly from planet-lab.eu and output a node input file

In the *deployment* folder you will find a Python script that does this: *planetlabgetnodes.py*

Edit this file, locate the Python dictionary that is depicted below and add your planet-lab.eu username and password:

```
auth = { 'AuthMethod' : 'password',
         'Username' : '<planetlab user name>',
         'AuthString' : '<planetlab user passwd>',
}
```

This file will access planet-lab.eu, retrieve the sliver hostnames that you have allocated, resolve their IP addresses and produce the so-called node input file with name *planetlab.cfg* which has the following format:

```
network = {
    nodes = (
    {
        testbed_ip = "193.63.75.18";
    },
    {
        testbed_ip = "128.232.103.201";
    },
    {
        testbed_ip = "195.148.124.73";
    }
    );
};
```

Before you invoke the deployment tool, the generated file should be enriched with the following information (which should be placed at the top of the file, also replace appropriate fields with what applies in your case):

```
BLACKADDER_ID_LENGTH = 8;
LIPSIN_ID_LENGTH = 8;
CLICK_HOME = "/home/certhple_purs1/";
WRITE_CONF = "/tmp/";
USER = "certhple_purs1";
SUDO = true;
OVERLAY_MODE = "ip";
```

ii) Transfer a .tar.gz bundle and automatically decompress it at home folders on all slivers

In case you need to transfer Click, Blackadder and other application binaries, you can appropriately create a .tar.gz bundle of them. Keep in mind that it has to be a .tar.gz because tar zxvf is the command that will be issued remotely for decompression and everything gets transferred and decompressed at the home folders on the PlanetLab slivers, so the home folder will be the root reference. The deployment tool accepts a flag, *filename* as follows: *-t <tgzfile>* and commences file tranfer and remote decompression before it deploys the blackadder instances. This mode of operation is, of course, optional during deployment.

iii) Autogenerate graphs from a node input file using the Barabasi-Albert model for scale-free networks. The autogenerated graph is saved in standard deployment tool configuration file format and can be re-used in experiments.

Doing this is optional and depends on the experiment needs. The user may directly define a standard configuration file format and use that on PlanetLab, rather than following this approach. If you have

allocated many PlanetLab nodes and you need an autogenerated graph then you may use this approach. For this to work, the deployment tool expects a node file format as described in (i). This file contains all the PlanetLab nodes along with global configurations. The deployment tool should be invoked with the *-c <node configfile> -a* flag combination. *-c* designates the configuration file name and *-a* lets the tool know that the configuration file format has plain nodes without connections and that the tool should use Albert-Barabasi model to define connections.

The tool finds the node that has the least amount of total hops from the rest of the nodes on the autogenerated graph and sets this as the Rendezvous and TM node. You can always change that in the autogenerated file.

iv) Build and distribute Click configuration files as well as remotely start Blackadder instances in testbed nodes

This is typical behavior in a testbed that the tool will always do that as a result of any proper configuration. The user does not have to define extra parameters for this.

Running Example Applications

Samples

Several sample applications are included in Blackadder's release.
In *examples/samples*, run make to compile these applications.

There are also simple examples for the SWIG-based Python, Ruby and Java bindings in language-specific directories under *examples/swig-samples*. Similar examples for the C binding can be found in the *examples/c-samples* directory.

Video Streaming

C++ version

Run *make* in ~/examples/video_streaming
VLC is required to run this example application.
Video Publisher: run *./video_publisher*
Using the console-based interface, advertise videos by typing the exact name of each video.
Advertise the catalogue of videos by typing . and then press enter.
Video Subscriber: run *./video_subscriber*
Upon initiation the subscriber subscribes to the video catalogue and receives it by the publisher.
Then, type the exact name of one of the advertise videos to join the video stream.

Java version

The Java version of the video streamer provides a graphical user interface for both the publishing and subscribing to video and media streams. Using the graphical version of the publisher, we can currently publish video files and audio files. We can also publish Internet media streams such as Internet radio and YouTube media (although some YouTube media will not currently work).

There are no particular requirements for running the demo although there are some things you'll need to do - I'll assume a blank install here on something like Debian:

1) Install VLC (luggage or later should be ok).

2) You will need to build the Java bindings (instructions are included in the Java bindings directory - lib/bindings/java-binding). Make sure you get the correct location of your jni.h file in the Makefile and then do a 'make'.

3) Edit the project.properties file and correct the path to your specific

"java-binding/jni/eu_pursuit_client_BlackadderWrapper.o" file.

4) You must add the necessary jar files to your lib folder if they are not already there.

- The Apache Commons Codec
  - http://archive.apache.org/dist/commons/codec/binaries/commons-codec-1.5-bin.zip
- The JNA libraries
  - http://java.net/projects/jna/sources/svn/content/tags/3.3.0/jnalib/dist/jna.jar
  - http://java.net/projects/jna/sources/svn/content/tags/3.3.0/jnalib/dist/platform.jar
- The VLCj jar
  - http://vlcj.googlecode.com/files/vlcj-1.1.5.1.jar

5) In the video demo directory, type 'ant' to build the project. You may need to edit the build.xml file and change line 9 to point to your Blackadder location.

6) You need to start blackadder. For example, by running "sudo click sample.conf" in the blackadder src directory.

7) You're now ready to run the app! The two commands are:

ant publisher

ant subscriber

Run the publisher first, add some videos and then run the subscriber. You will need to refresh the catalog whenever you change it in the publisher. The video should start when you click on subscribe - it should stop when you unsubscribe. If you then re-subscribe, it should start again from the beginning.

You should **not** subscribe to more than one video at a time (as a single user). To subscribe to another video, you should unsubscribe from the previous one first. Also, please note that this is a demo and is likely to crack under pressure....

Voice over Blackadder

The Java-based voice application is located in *{blackadder_dir}/examples/java-voice*. The application uses the non-SWIG Java binding, therefore, before running the application, you need to compile and build the respective binding following the instructions above.

Once you have the Java binding built, track the location of the binding object file (*eu_pursuit_client_BlackadderWrapper.so*). Edit the *config.txt* file located in the java-voice directory and set the *JNI_PATH* variable to point to the location on the Java binding's object file.

You also need to download and add jar files for *Visual Swing for Eclipse* into the *java-voice/lib* folder if they are not already there. The file to download and to extract is:
http://visualswing4eclipse.googlecode.com/files/vs4e_0.9.12.I20090527-2200.zip

To build the Java sources just run "*ant*" (*ant* is also required to build the Java binding's Java sources). In case you modify the Java binding itself, make sure to copy the respective jar in the application's *lib* folder.

To start the application, first start the Blackadder network and then run

```
./run-application.sh -f conf_file
```

with "*conf_file*" pointing to the location of you *config.txt*.

One the pop-up window, choose whether you want to make the call or wait for incoming calls. One of the two participants has to "Make call" (caller), providing his/her name and the callee's name as well. Then presses the "Call" button. The other participant awaits for the call (callee). Press "Receive call", enter you name (it has to match the callee name set by the caller) and accept the call.

Running an NS3 Simulation

Before running an NS3 Simulation a configuration file that describes the simulated network topology must be created. This file describes all network nodes and connections as well as the applications that will run in each node.

Such sample configuration file (called ns3_topology1.cfg) exists in

BLACKADDER_PREFIX/ ns3/blackadder-model/model

Deploy a simulated network by running:

./deploy -c BLACKADDER_PREFIX /ns3/blackadder-model/model/ns3_topology1.cfg –s

This will create files in /tmp/:

00000001.conf, 00000002.conf, 00000003.conf, 00000004.conf, 00000005.conf, 00000006.conf, 00000007.conf, 00000008.conf, topology.graphml, topology.cpp

Leave all of them there except topology.cpp. In topology.cpp add the following includes (in general if a new NS3 application is written and included in the simulated topology, the respective header files must be **manually** included. Here we use two applications called Publisher and Subscriber):
#include "publisher.h"
#include "subscriber.h"

cp /tmp/topology.cpp <NS3_PREFIX>/ns-3.15/examples/blackadder/example3.cc

*Now **edit** <NS3_PREFIX>/ns-3.15/examples/blackadder/wscript to add the example (it will be called example/blackadder/example3 in ns3):*

obj = bld.create_ns3_program('example3', ['core', 'point-to-point', 'blackadder', 'applications'])
obj.source = ['example3.cc', 'publisher.cc','subscriber.cc',]

Now build ns3:
cd <NS3_PREFIX>
./waf build
You can run the example simulation with
./waf --run examples/blackadder/example3

Writing an NS3 Application

All NS3 pub/sub applications running on top of a simulated Blackadder deployment must extend the PubSubApplication Class. Since pub/sub applications are NS3 Objects, they should implement the GetTypeId(void) method, where they can define their own attributes (see NS3 attribute system).

Four methods must be implemented:

DoStart(void): This is a good place to define the Event Listener Callback method that will be called when a Blackadder is pushed to the application by the simulator (see Publisher and Subscriber example applications). A call to *PubSubApplication::DoStart();* must be also made so that some housekeeping is done in the father class.

DoDispose(void): This is a good place to deregister and free any resources acquired with the DoStart method. A call to *PubSubApplication::DoDispose();* must be also made.

StartApplication(void): This method is called by the Simulator when the time has come for the application to start (defined in the deployment configuration file). The API exported by the PubSubApplication Class can be used to access the service model which is exported by Blackadder. NS3 events can be also scheduled .

StopApplication(): This method is called by the Simulator when the time has come for the application to stop (defined in the deployment configuration file). Any pending events must be cancelled. Blackadder is notified about the application exiting, by the underlying ServiceModel Class that hides such implementation details from a NS3 application.

Finally, an event handler must be defined and assigned to the *m_cb* variable of PubSubApplication object. This should happen in the DoStart() method call, like:

m_cb = MakeCallback(&ApplicationClassName::EventHandler, this);

The event handler's signature must be as following:

void ApplicationClassName::EventHandler(Ptr<Event> ev);

It should usually look like this:

```
void ApplicationClassName::EventHandler(Ptr<Event> ev) {
    switch (ev->type) {
      case SCOPE_PUBLISHED:
        //do something – maybe now or schedule an NS3 event
        break;
      case SCOPE_UNPUBLISHED:
        //do something – maybe now or schedule an NS3 event
         break;
      case START_PUBLISH:
         //do something – maybe now or schedule an NS3 event
        break;
      case STOP_PUBLISH:
         //do something – maybe now or schedule an NS3 event
        break;
      case PUBLISHED_DATA:
         //do something – maybe now or schedule an NS3 event
        break;
    }
  }
```